

Jolie Community on the Rise

Alexey Bandura*, Nikita Kurilenko*, Manuel Mazzara*, Victor Rivera*, Larisa Safina*, Alexander Tchitchigin*[†]

*Innopolis University, Russia

{a.bandura, n.kurilenko, l.safina, a.chichigin, m.mazzara, v.rivera}@innopolis.ru

[†]Kazan Federal University, Russia

a.tchichigin@it.kfu.ru

Abstract—Jolie is a programming language that follows the microservices paradigm. As an open source project, it has built a community of developers worldwide - both in the industry as well as in academia - taken care of the development, continuously improved its usability, and therefore broadened the adoption. In this paper, we present some of the most recent results and work in progress that has been made within our research team.

I. INTRODUCTION

Monolithic applications have long been widespread. They are applications composed of modules that are not independent from the application to which they belong since they share resources of the same machine (e.g. memory, files). Hence, monolithic applications are not dependently executable, bringing with them a series of drawbacks, e.g. scalability and maintainability: a change to a small part of the application requires the entire monolith to be rebuilt and redeployed. Plugging together components the same way that construction engineering does, component-based development appeared as a much more flexible approach. To some extent, “Microservice Architecture” [12] is the extreme of this run towards componentization observed over the last few decades. It differs from a typical Service-Oriented Architecture in terms of a service size, bounded context and independency [8]. A microservice combines related functionalities into a single business unit implemented as a minimal independent process interacting with other microservices via messages. So, instead of having a monolithic application built as a single unit, the system is built as a set of small autonomous services that are independently deployable, have firm module boundaries, and can be implemented using a variety of programming languages and technologies. These services run their own processes and use lightweight communication mechanisms. The term describing the new architectural style has been coined in recent years and has attracted lots of attention as a new way to structure applications.

The Jolie programming language [21], [4] was created in order to maximize the use of the microservices architectural style. In Jolie, microservices are first-class citizens: every microservice can be reused, orchestrated, and aggregated with others [19]. This approach brings simplicity in components management, reducing development and maintenance costs, and supporting distributed deployments [9].

The development of Jolie followed a major formalization effort for workflow and service composition languages, and the EU Project SENSORIA [2] has successfully produced a

plethora of models for reasoning about composition of services (e.g., [13], [14], [15]). On the mathematical side, the formal semantics of Jolie [11], [10], [18] have been inspired by different process calculi, such as CCS [16] and the π -calculus [17]. From a practical point of view, however, Jolie is a descendant of standards for Service-Oriented Computing such as, for example, WS-BPEL [6]. With both theoretical and practical influences, Jolie is a suitable candidate for the application of recent research techniques, e.g., runtime adaptation [23], process-aware web applications [20], or correctness-by-construction in concurrent software [7].

Our research team published several results to this regard. Recent works were devoted to the extension of Jolie type system [24] [25], showing how computation can be moved from a process-driven to a data-driven approach by means of introducing a choice type. This paper presents recent developments for the Jolie language carried out by our research team at Innopolis University. The paper is structured as follows. In Section II, we briefly review the Jolie programming language in order to simplify the understanding of the forthcoming sections. Section III presents an implementation of a new version of the **foreach** operator that seeks to simplify the iteration over data structures in Jolie programs. Section IV discusses the implementation of an inline documentation for Jolie developed under the form of an **Atom** package. Finally, Section V draws some conclusions and looks at the future.

II. JOLIE PROGRAMMING LANGUAGE

Each Jolie program consists of two parts: behavior and deployment.

The deployment part contains directives that help the Jolie program to receive and send messages and be orchestrated among other microservices. Such directives include:

- *Interfaces*: sets of operations equipped with information about their request (and sometimes response) types
- *Message types*: can be represented as native types, linked types, or undefined
- *Communication ports*: define how communications with other services are actually performed

The deployment part is separated from the program behavioral part, so that the same behavior may be reused later with a different deployment configuration.

The behavioral part defines the microservice implementation, containing both computation and communication expressions. Each behavior part has a **main** procedure that defines an entry

point of execution. Activities performed by the service can be composed sequentially, in parallel, or with (input guarded) non-deterministic choice [21]. Execution of behavior part is made by means of standard control flow statements, like conditional operators, count-controlled loops and collection-control loop (**foreach** operator) which we will discuss in more details in Section III.

Communications in Jolie are type-checked at run-time when a server receives a message [19], [22]. Message types are introduced in the deployment part of Jolie programs by means of the keyword **type** followed by the type identifier and definition. Type definition can be native, native with subtypes, native with undefined subnodes, link type, or can be undefined (meaning that variable is null until a value is assigned to it).

Types may have any number of subtypes, also known as subnodes for its tree-like representation. For example, we can define the following structure path:

```
europe.ireland.city = ‘Dublin’
```

The structure has a root node “europe” and a subnode “ireland”, which has a subnode “city”. Each node has a cardinality. If it is not specified, then it defaults to one (meaning that there is only one entity).

III. EXTENDING THE SEMANTICS OF **FOREACH** LOOPS IN JOLIE

In the Jolie language, to iterate over the values of a variable, it is necessary to specify the path where the node is. For instance, the follow code iterates over the values of variable `a.b`:

```
for(i = 0, i < #a.b, i++){
  printlnConsole(a.b[i]); }
```

The aid `#` determines the number of values in a specific node and `a.b[i]` accesses the *i*th value of subnode `a.b`. Jolie does not restrict the user on the length of the variable that he can use. Hence, iterating over longer variables nodes (e.g. `a.b.c.d.e` or `a.b.c.d.e.f.g`) leads to cumbersome loop body: firstly, it is needed to type the full variable path every time an access to a specific values is needed. This is a time-consuming task, especially when variable paths are too long; secondly, the code uses a counter to iterate over the array. Hence, it is prone to error. One needs to continually pay attention to the number of items in the array while also setting up the index itself.

Jolie offers a workaround to improve the previous code making it less cumbersome. In Jolie, one variable path can be aliased to a variable, meaning that a long variable path can be expressed in terms of a short one. Aliases are created with the `->` operator, for instance the code

```
var1 -> a.b.c.d[1];
var2 -> a.b.c
```

aliased the path variable `a.b.c.d[1]` to `var1` and `a.b.c` to `var2`. So, the previous example could be rewritten as

```
var -> a.b;
for(i = 0, i < #a.b, i++){
  printlnConsole(var[i]);
}
```

Even though the code might be clearer and more readable, aliasing in Jolie does not bring any performance improvement and the user still needs to deal with indexes.

Another workaround to tackle this problem is to use the **foreach** operator defined in Jolie. **foreach** is defined to transverse Jolie data structures. The syntax is

```
foreach (nameVar : root){ //code block }
```

The **foreach** operator will iterate over the nodes defined in `root` executing the internal code block at each iteration. The main issue with this workaround is that it will iterate over the root node of a path variable. The code

```
foreach (nameVar : a.b){ //code block }
```

will iterate over `a` (as it is the root of `a.b`) instead of over the subnode `a.b`.

The proposed **foreach** seeks to follow a more natural scheme: “iteration through all elements in a specific (sub)node” while hiding to users the heavy burden of dealing with indexes. The syntax and semantics of the current **foreach** operator is extended following similar semantics as the ones found in other programming languages, e.g. C++ or Java.

A. Extending the **foreach** loop in Jolie

As stated before, the current Jolie syntax does not provide any mechanism to transverse a (sub)node in the tree path of a variable. The proposed extension for the **foreach** loop will promote clarity and usability of the language, enabling users to have a more structured source code.

The extension comprises two current Jolie constructs: **foreach** loops and alias syntax: on one hand, the **foreach** loop will keep an implicit index that will be managed by the language, hiding the heavy burden to the users; and another hand, it will use alias syntax eliminating clumsy code brought by common long variable paths, while encapsulates all low-level details that are irrelevant in our context.

The following shows the syntax extension of the **foreach** construct:

```
foreach (lVariablePath -> rVariablePath){
  //code block using lVariablePath
}
```

where `lVariablePath` is a fresh variable aliased to the variable path `rVariablePath`. `rVariablePath` needs to represent a node in the variable path rather than a specific value. The **foreach** operator will iterate over the values defined in the node `rVariablePath` executing the code block at each iteration.

The implementation is a syntax sugar for the **for** loop presented previously. It will, internally, translates the previous code to:

```
for (i = 0, i < #rVariablePath, i++){
    lVariablePath -> rVariablePath[i];
    //code blockusing lVariablePath
}
```

The **foreach** extension was implemented by implementing a visitor to transverse Jolie’s code AST and generating the previous Jolie code. The implementation can be found at [3]

Having this new syntax and semantics for **foreach**, users can iterate over long path variables without introducing cumbersome code or having the heavy burden of dealing with indexes.

Compared to Java where an enhanced for-loop can’t be used to remove or update elements as you traverse a collection, **foreach** in Jolie are capable to do this:

```
foreach (var -> node.subnode){
    var = something
}
```

The last point is that Jolie **foreach** employs an alias sign “->” instead of a colon symbol used in Java and C++ to separate a target collection and an iteration variable. The colon symbol in Jolie is already used to traverse the subnodes of a particular node (e.g. `subnode_1`, `subnode_2`, ... `subnode_n`) while we are iterating through the values of the specific node (e.g. `node[0]`, `node[1]`, ... `node[n-1]`). Additionally, by employing “->” we explicitly demonstrate that **foreach** uses an alias syntax while traversing values of a node.

IV. DISPLAYING INLINE DOCUMENTATION FOR JOLIE

Commonly, developers prefer the use of Integrated Development Environments (IDEs) since they provide a vast amount of functionalities within the same context. They come, for instance, with the corresponding compilers or interpreters, testing and debugging tools, code navigation and autocompletion, and so on.

Jolie’s IDE is based on Atom [1] text editor, which is a powerful open-source editor. An advantage of using an IDE based on Atom, is that one can extend its functionality by implementing additional packages and coupling them to the main core.

Jolie’s IDE is still under work in progress. For instance, the IDE does not support the invoking of a definition of a type or a port (when the cursor is standing above an identifier) as an inline small window to recall the programmer of it. This section shows the development of a feature for displaying inline documentation in Jolie’s IDE for Jolie’s code. The inline documentation view was implemented as an Atom’s package. The package shows in a small window brief information (the inline documentation) about the subject which is selected with the cursor (see Figure IV.1). The idea behind this feature is for the programmer to have a quick view of the documentation of a specific identifier without the need of opening the file where the documentation is.

A. Deployment part of Jolie

As stated in Section II, Jolie programs consist of two parts: behavior and deployment. The support for inline documentation takes into account the deployment part.

In the deployment part, the communication is supported by communication ports. They define the communication links used by services to exchange data. There exist two kinds of ports: input and output ports and their syntaxes are similar. Ports are based upon the three fundamental concepts: **Location**, **Protocol** and **Interface**. The following is the syntax for an output port in Jolie:

```
outputPort id {Location: URI
                Protocol: p
                Interfaces: iface_1}
```

`id` is the name of the port. **Location** expresses the communication medium, `URI` stands for Uniform Resource Identifier, and it defines the location of the port. The **protocol** defines how data needs to be sent or received. It is defined by `p`. Finally, **Interfaces** are the interfaces accessible through the port. The implementation of the inline doc view will give information about the kind of protocols (defined in **protocol**) and the interfaces (defined in **Interfaces**)

B. Implementation

The inline documentation for Jolie (herein *InlineDoc View*) was implemented as a plugin of Jolie’s IDE (i.e. a package of Atom). When the user clicks on the word he wants to see the documentation, a window appears showing the corresponding inline documentation, as depicted in Figure IV.1. The window gives a snippet of the documentation and presents two buttons, “docs” and “online”. “docs” gives the user the possibility to show the documentation ...; the “online” button translates the representation of the documentation to HTML code so the documentation can be showed in a browser.

The plugin implements a series of components that interact to each other sequentially. When the user clicks on a word in the Jolie IDE, one of the predefined Atom’s components retrieves the string text corresponding to the line where the cursor is on. This information is sent to a pre-process component. This component

- 1) filters the line: checks whether the user clicked on a word that might contain documentation. The implementation of the *InlineDoc View* takes into account only the deployment part of Jolie programs. If users click on something different, no information is retrieved; it also checks whether the word has some meaning, e.g. it is not a curly bracket or a Jolie keyword, from which no information is available.
- 2) categorizes the type of word the user clicked on. The word can correspond to the protocol or to the interface parts of the port communication.

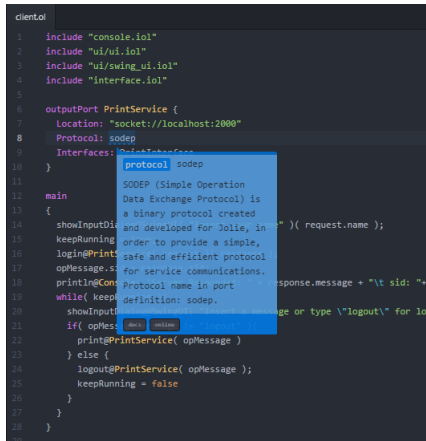


Figure IV.1. Inline documentation for Jolie code

This component sends both the corresponding *word* (which correspond to a word that some information can be retrieved from) and the *categorization* (either the word is defined in the protocol or in the interface part) to the evaluation component. This component will search for the documentation of the *word* in the *categorization* files. Categorization files are JSON files provided by us that define a mapping between protocols and documentation. This component will send the corresponding documentation to another Jolie IDE component that is in charge of displaying the text: the InlineDoc View window. The information received by this component uses markdown format [5]. Markdown is a text-to-HTML conversion tool that enables users to get a structured valid XHTML (or HTML) from a text. So this component can also display the information (upon request of the user by clicking on the “online” button) in a browser.

V. CONCLUSIVE REMARKS

Research on microservices and Jolie is progressing and several scientists have noticed the interesting features and the challenges connected with language development. This paper reports our latest results and highlights realization details. Two aspects have been covered here and represent the contribution of our work:

- *Foreach operator*: we presented an implementation of a new version of the **foreach** operator aiming at simplifying the iteration over data structures in Jolie programs. It is syntactic sugar, but makes non-trivial programs more straightforward to write and read.
- *Inline documentation* : we introduced the implementation of an inline documentation for Jolie developed under the form of an Atom package. Jolie’s IDE is work in progress, and much as to be done yet. Still, inline documentation supports developers in writing programs and ultimately enhance code quality.

Open challenges are related to the static type checking of Jolie, a feature that is necessary for a broader adoption of the

language in an industrial context. The principal effort of our research in the upcoming months will therefore focus on static analysis.

ACKNOWLEDGEMENTS

The authors would like to thank Daniel Johnston for assistance. Fabrizio Montesi played an important role in providing us with interesting challenges and following the technical execution. In particular, the **foreach** extension comes from his specification, and he also contributed to the final realization. The refinement and stabilization of this extension for release is now under way.

REFERENCES

- [1] Atom. Text editor. <https://atom.io/>.
- [2] EU Project SENSORIA. Accessed April 2016. <http://www.sensoria-ist.eu/>.
- [3] GitHub. Online project hosting using Git. <https://github.com/jolie/jolie/commit/0d21cb84fc96cf222bc8442ece6282086a890993>.
- [4] Jolie Programming Language. Accessed April 2016. <http://www.jolie-lang.org/>.
- [5] Markdown. Markup language. <http://daringfireball.net/projects/markdown/syntax>.
- [6] WS-BPEL OASIS Web Services Business Process Execution Language. accessed April 2016. <http://docs.oasis-open.org/wsbpel/2.0/wsbpel-specification-draft.html>.
- [7] Marco Carbone and Fabrizio Montesi. Deadlock-freedom-by-design: multiparty asynchronous global programming. In *POPL*, pages 263–274, 2013.
- [8] N. Dragoni, M. Mazzara, S. Giallorenzo, F. Montesi, A. Luch Lafuente, R. Mustafin, and L. Safina. Microservices: yesterday, today, and tomorrow. <https://arxiv.org/pdf/1606.04036.pdf>, (2016).
- [9] M. Fowler. Microservice Trade-Offs. <http://martinfowler.com/articles/microservice-trade-offs.html>, (2015).
- [10] C. Guidi, I. Lanese, F. Montesi, and G. Zavattaro. Dynamic error handling in service oriented applications. *Fundam. Inform.*, 95(1):73–102, 2009.
- [11] C. Guidi, R. Lucchi, G. Zavattaro, N. Busi, and R. Gorrieri. Sock: a calculus for service oriented computing. In *ICSOC, volume 4294 of LNCS*, pages 327–338. Springer, 2006.
- [12] J. Lewis and M. Fowler. Microservices. Accessed September 2015. <http://martinfowler.com/articles/microservices.html>, (2014).
- [13] R. Lucchi and M. Mazzara. A pi-calculus based semantics for WS-BPEL. *J. Log. Algebr. Program.*, 70(1):96–118, 2007.
- [14] M. Mazzara, F. Abouzaid, N. Dragoni, and A. Bhattacharyya. Toward design, modelling and analysis of dynamic workflow reconfigurations - A process algebra perspective. In *Web Services and Formal Methods - 8th International Workshop, WS-FM*, pages 64–78, 2011.
- [15] Manuel Mazzara. *Towards Abstractions for Web Services Composition*. PhD thesis, University of Bologna, 2006.
- [16] R. Robin Milner. *A calculus of communicating systems*. Lecture notes in computer science. Springer-Verlag, Berlin, New York, 1980.
- [17] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, I and II. *Information and Computation*, 100(1):1–40, 41–77, September 1992.
- [18] F. Montesi and M. Carbone. Programming Services with Correlation Sets. In *Proc. of Service-Oriented Computing - 9th International Conference, ICSOC*, pages 125–141, 2011.
- [19] Fabrizio Montesi. JOLIE: a Service-oriented Programming Language. Master’s thesis, University of Bologna, 2010.
- [20] Fabrizio Montesi. Process-aware web programming with Jolie. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing, SAC '13*, pages 761–763, New York, NY, USA, 2013. ACM.
- [21] Fabrizio Montesi, Claudio Guidi, and Gianluigi Zavattaro. Service-oriented programming with jolie. In Athman Bouguettaya, Quan Z. Sheng, and Florian Daniel, editors, *Web Services Foundations*, pages 81–107. Springer, 2014.
- [22] J. M. Nielsen. A Type System for the Jolie Language. Master’s thesis, Technical University of Denmark, 2013.
- [23] Mila Dalla Preda, Saverio Giallorenzo, Ivan Lanese, Jacopo Mauro, and Maurizio Gabbriellini. AIOJ: A choreographic framework for safe adaptive distributed applications. In *Software Language Engineering - 7th International Conference, SLE 2014, Västerås, Sweden, September 15-16, 2014. Proceedings*, pages 161–170, 2014.

- [24] Larisa Safina, Manuel Mazzara, Fabrizio Montesi, and Victor Rivera. Data-driven workflows for microservices (genericity in jolie). In *Proc. of The 30th IEEE International Conference on Advanced Information Networking and Applications (AINA)*, 2016.
- [25] Alexander Tchitchigin, Larisa Safina, Manuel Mazzara, Mohamed Elwakil, Fabrizio Montesi, and Victor Rivera. Refinement types in jolie. <https://arxiv.org/pdf/1602.06823.pdf>.